

第2章

マルチプロセッサに対応したリアルタイムOSとは

――機能分散型マルチプロセッサ向けに拡張されたμITRON 4.0仕様

本田晋也

組み込みシステム構築の基盤となる各種ソフトウェアの開発と普及推進を目的とする業界団体「TOPPERSプロジェクト」は，マルチプロセッサに対応したリアルタイムOS「TOPPERS/FDMPカーネル」を開発した．ここでは，従来のシングル・プロセッサ向けのμITRON仕様と異なる点を中心に，FDMPカーネルの概要を紹介する．　（編集部）

ここ1，2年の間に，複数のCPU（もしくはDSP）を一つのチップに搭載した，いわゆるマルチコアLSIが数多く発表されています．米国Intel社の「Pentium」，米国Advanced Micro Devices（AMD）社の「Opteron」や「Athlon」，米国IBM社の「PowerPC」，米国Sun Microsystems社の「UltraSPARC」など，パソコン用やワークステーション用の主だったプロセッサでは，マルチコア化した製品が用意されています．

これらのプロセッサがマルチコアとなっている理由は二つあります．一つは並列性が必要とされる分野からの要求です．例えば，科学技術計算やサーバの用途では，複数のプロセッサを用いたマルチプロセッサ構成により処理能力を向上させる手法が一般的に用いられてきました．この用途では，一つの高速なプロセッサよりも比較的動作周波数の低い複数のプロセッサを用いたほうがメリットのある場合があります．また，マルチコアのプロセッサを用いることにより，より小さい設置面積でより多くの機能を実装できるというメリットもあります．

もう一つは動作周波数の限界です．パソコン用やワークステーション用のプロセッサはこれまで動作周波数を引き上げることで性能を向上させてきました．しかし，ここに来て動作周波数の向上に限界が来ています．そこで，プロセッサの性能（価値）を上げる手段としてマルチコア化が利用されています．

1 組み込み機器における マルチプロセッサの利用

一方，組み込み機器の分野でも多くのマルチコアLSIが登場しています．例えば，米国Texas Instruments社の「OMAP」や東芝の「MeP」，英国ARM社の「MPCore」，NECエレクトロニクスの「MP211」などです．

組み込み機器の分野でマルチコアが用いられている要因として，消費電力の問題が挙げられます．組み込み機器には，携帯電話や携帯型オーディオ機器のようにバッテリで駆動するものが多く，消費電力の改善がそのまま商品価値につながります．この消費電力に着目すると，高速なプロセッサを一つ用いるよりも，低速なプロセッサを複数用いたほうが有利になります．

別の要因として，専用LSIによる処理の固定化の問題があります．これまで，プロセッサを利用する場合でも，負荷の重い処理は専用LSIで扱っていましたが，専用LSIは機能が固定されるため，仕様の変更には柔軟な対応ができないという欠点があります．そこで，それらの機能を複数のプロセッサで実現することにより，機能に柔軟性を持たせようという流れがあります（例えば，MePなど）．

また，マルチメディア関連の処理は，汎用CPUよりもDSPのほうが高速かつ効率良く処理できるため，汎用CPU＋DSPという非対称の構成をとるものが多くなっています（例えば，OMAPなど）．

● マルチプロセッサのタイプ

ここで，本稿におけるマルチプロセッサの構成を定義し



ておきます．

対称マルチプロセッサ（SMP）は，アプリケーション・ソフトウェアから見ると，どのプロセッサも対等に見えます．この構成はパソコンやワークステーションなどで一般的に用いられています．こうしたシステムでは，複数のプロセッサを用いて，依存性のない複数の処理（例えば，サーバへのサービスの要求に対する処理など）を並列に実行します．どのような処理を実行するかは，システムの実行状態にならなければわからないため，処理（タスク）が発生するたびにOSがどのプロセッサで実行するかを決定します（図1）．

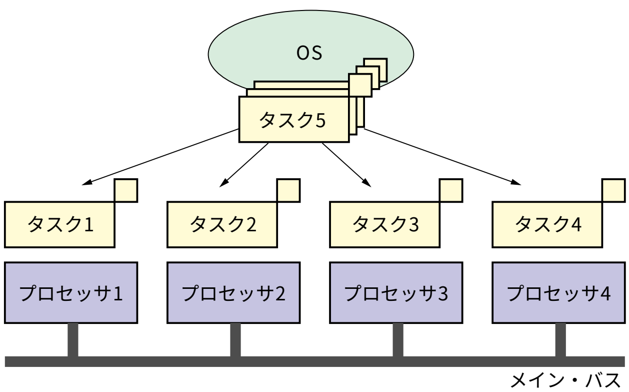
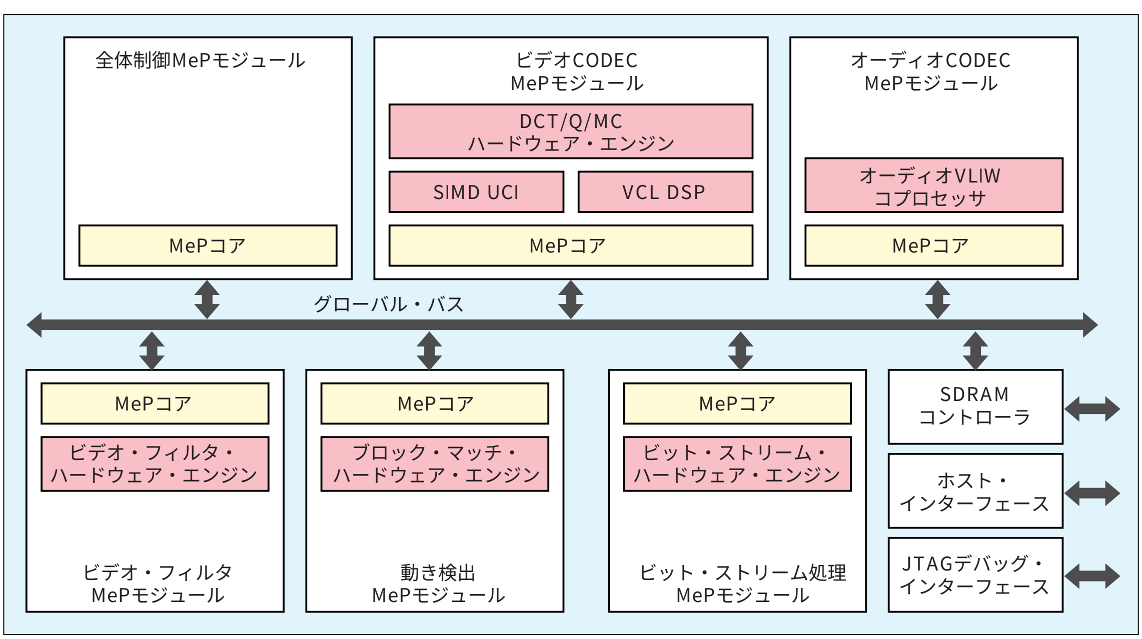


図1　対称マルチプロセッサの構成

処理（タスク）が発生するたびに，OSはそのタスクを動的にプロセッサに割り当てる．

一方，組み込みシステムの場合，パソコンなどと異なり実行する処理が事前にわかっているケースが一般的であるため，機能分散型のマルチプロセッサ・システムを用いることが多いと言えます．機能分散型マルチプロセッサ・システムは，プロセッサごとに実行する処理が決まっています．そのため，各プロセッサの周辺回路の構成が異なったり，前述の汎用プロセッサ＋DSPのようにプロセッサ・コアの種類が異なる場合が多いようです．なお，ここでは，ハードウェア的に同じものであっても，プロセッサごとに実行する処理が異なっている構成も「機能分散型」と呼ぶことにします．

機能分散型マルチプロセッサの例として，図2に東芝のMePを用いたMPEG-2 CODEC LSIを示します．各プロセッサ・モジュールは，アプリケーションに応じて最適化されており，プロセッサ・コアであるMePコアと専用ハードウェアによって構成されています．そして，それぞれのモジュールで行う処理に応じてコアや専用ハードウェアの構成が異なります．



DCT：離散コサイン変換　Q：量子化　MC：動き補償　UCI：ユーザ・カスタム命令
SIMD：single instruction multiple data　VLC：variable length coding　VLIW：very long instruction word

図2　機能分散型マルチプロセッサの例

機能分散型マルチプロセッサの典型的な例として，東芝のMePを用いたMPEG-2 CODEC LSIを示す．このLSIは六つのMePモジュールから構成されている．各MePモジュールの機能は固定されており，それぞれプロセッサ・コアであるMePコアと割り当てられた機能のための専用ハードウェアを内蔵している．

● 現状のソフトウェア開発とOSの必要性

これまで機能分散型マルチプロセッサのソフトウェア開発では，各プロセッサにシングル・プロセッサ用のリアルタイムOSを載せていました(OSを利用しないこともある)．そしてプロセッサ間の同期・通信は，リアルタイムOSの機能を使わず，アプリケーション・レベルで実現していました(シングル・プロセッサ用OSがプロセッサ間の同期・通信機能を持たないため)．この方法では，アプリケーションごとにプロセッサ間の同期・通信機能を実現する必要があるため，開発工数が増えてしまいます．また，ある処理を別のプロセッサに移そうとした場合，同期・通信部分の作り直しが必要になります．

そこで筆者ら(TOPPERSプロジェクト)は，プロセッサ間の同期・通信をリアルタイムOSで実現することにより，ソフトウェア開発の効率を上げることを考えました．機能分散型マルチプロセッサ・システムは応用ごとにその構成が最適化されるため，ハードウェア的なプロセッサ間の同期・通信方法もさまざまな形態をとる可能性がありますが，リアルタイムOSを用いることにより，この違いを吸収することもできます．

マルチプロセッサ対応のリアルタイムOSが提供するプロセッサ間の同期・通信に，シングル・プロセッサ用のOSが持つタスク間の同期・通信との互換性があれば，ソフトウェア開発者はプロセッサの違いを意識することなくプログラミングできることになります．また，シングル・プロセッサ用のアプリケーションの移植もスムーズに行えます．

2 マルチプロセッサ対応μITRON仕様

機能分散型マルチプロセッサ向けのリアルタイムOSの必要性から，筆者らはμITRONの拡張仕様を策定し，その拡張仕様を「TOPPERS/FDMPカーネル(以下，FDMPカーネル)」として実装しました．なお，実装については，「2004年度の未踏ソフトウェア創造事業」の支援を受けました．以下に，拡張仕様について解説します(実装については本特集の第3章で解説する)．

● できるだけ特別な機能は持たせない

FDMPカーネルは，μITRON 4.0仕様の標準プロファイル注をベースに策定しました．拡張にあたっては，機能分散型マルチプロセッサ独自の機能，すなわちAPI(application program interface)やシステム・コール(プロセスからの機能の呼び出し規約)は，極力設けない方針としました．

これは前述したように，プロセッサ間の同期・通信とタスク間の同期・通信に互換性を持たせることにより，開発効率の向上やソフトウェアの移植を容易にするためです．

● カーネル・オブジェクトの扱い

タスクやセマフォ(複数のタスク間の同期をとるしくみ)などのカーネル・オブジェクトは，いずれかのプロセッサに属します．あるプロセッサに属するオブジェクトの集合を「クラス」と呼びます．カーネル・オブジェクトをどのプロセッサに所属させるかは，後で説明するコンフィグレーション・ファイルによって静的に決めます．

OSは，クラスをカーネル・オブジェクトの一つとみなし，1から始まる連続したID番号で識別します．FDMPカーネルの制御対象となるマルチプロセッサ・システムにおいて，以下に示すタスクなどの処理単位となるオブジェクトは，それが属するプロセッサでのみ実行されます．

- タスク
- タスク例外処理ルーチン
- 周期ハンドラ
- 割り込みハンドラ
- CPU例外ハンドラ

すなわち，あるタスクがどのプロセッサで実行されるかは静的に決定され，システム動作中にタスクを実行するプロセッサが変わることはありません．

タスクは，(属するプロセッサが異なっても)システムに存在するすべてのオブジェクトにアクセスできます．以下に示すカーネル・オブジェクトにはID番号が割り付けられています．

- タスク
- セマフォ
- イベント・フラグ
- データ・キュー
- メール・ボックス
- 固定長メモリ・プール
- 周期ハンドラ

注：μITRON 4.0仕様では，ソフトウェアの移植性を高めるため，標準プロファイルという機能セットを定めている．標準プロファイルでは，システム全体が一つのモジュールにリンクされ，カーネル・オブジェクトはすべて静的に生成される．



- ●タスクの起動
【C言語API】
ER ercd = act_tsk (ID tskid);
【パラメータ】
ID tskid 起動対象のタスクのID番号

- ●セマフォ資源の獲得
【C言語API】
ER ercd = wai_sem (ID semid);
【パラメータ】
ID semid 資源獲得対象のセマフォのID番号

図3 μITRONのAPIの例
act_tsk (ID tskid) では，tskidで指定したタスクを起動する．指定されたタスクは実行可能状態となり，システムのタスクの中で最高優先度ならばディスパッチして実行される．wai_sem (ID semid) では，semidで指定したセマフォの資源を獲得する．対象セマフォの資源がなければ呼び出したタスクは資源を獲得するまで待ち状態となる．

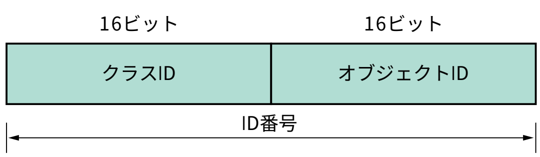


図4 オブジェクトIDの拡張
上位16ビットをクラスID，下位16ビットをクラス内のオブジェクトIDとする．

オブジェクトを操作するμITRON仕様のAPIは，引き数に操作対象となるオブジェクトのID番号を渡します(**図3**)．

機能分散型マルチプロセッサ仕様ではID番号を32ビットに拡張して，ID番号で所属するクラスも表します．具体的には，ID番号の上位16ビットで所属するクラスを，下位16ビットでそのクラス内におけるオブジェクトの識別番号を表します(**図4**)．上位ビットが'0'の場合，TCLS_SELFとして，そのAPIを発行したタスクと同じクラスに属するオブジェクトを対象とします．この拡張により，オブジェクトを操作するAPIを，そのままの形で機能分散型マルチプロセッサに対応させることができます．

● システム状態はそれぞれのプロセッサで独立している

シングル・プロセッサ用のカーネルが持つ以下のシステム状態は，機能分散型マルチプロセッサにおいては各プロセッサが持つことになります．

- ●実行状態のタスク
- ●CPUロック状態
- ●ディスパッチ(CPUの割り当て)禁止状態

つまり，シングル・プロセッサの場合と異なり，あるプロセッサでCPUロック状態やディスパッチ禁止状態になったとしても，ほかのプロセッサではそれらは禁止されていません．そのため，CPUロックやディスパッチ禁止を用いてほかのプロセッサの処理単位との排他制御は実現できません(**図5**)．

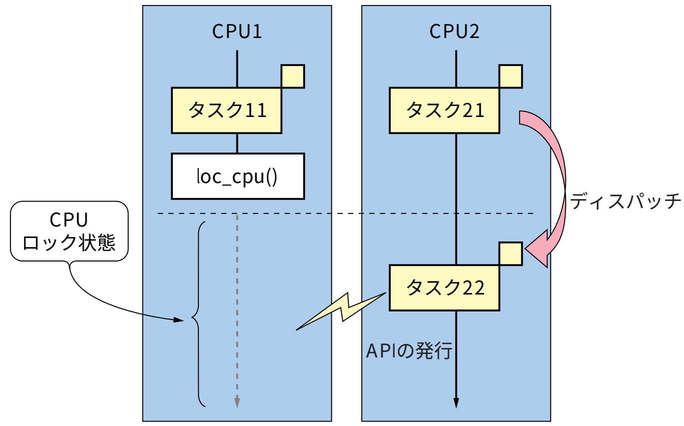


図5 プロセッサの状態
タスク11がCPUロック状態になるAPIを発行して，プロセッサCPU1がCPUロック状態になったとしても，プロセッサCPU2はその影響を受けず，ディスパッチを行ったり，タスク11に対してAPIを発行できる．

以下に示すシステム状態を変更/参照するAPIは，呼び出した処理単位が実行されているプロセッサに対して実行されます．

- ●get_tid/iget_tid (実行状態のタスクIDの参照)
- ●loc_cpu/iloc_cpu (CPUロック状態への移行)
- ●unl_cpu/iunl_cpu (CPUロック状態の解除)
- ●dis_dsp/ena_dsp (ディスパッチの禁止/許可)
- ●sns_loc (CPUロック状態の参照)
- ●sns_dsp (ディスパッチ禁止状態の参照)
- ●sns_dpn (ディスパッチ保留状態の参照)
- ●sns_tex (タスク例外処理禁止状態の参照)

※iが付いているのは非タスク・コンテキスト用

また，これらのAPIはとくにクラスIDを指定できるようには拡張していません．そのため，タスクは自分が属するプロセッサ以外のプロセッサの状態は変更できません．

● システム時刻はプロセッサごとに管理

システム時刻はプロセッサごとに管理します．カーネルは各プロセッサのシステム時刻の同期機能を持ちません．これは，システムの時刻の同期が必要な場合，ハードウェアで実現するほうが容易と考えたからです(例えば，単一のタイマの割り込みをすべてのプロセッサに入れる)．シ

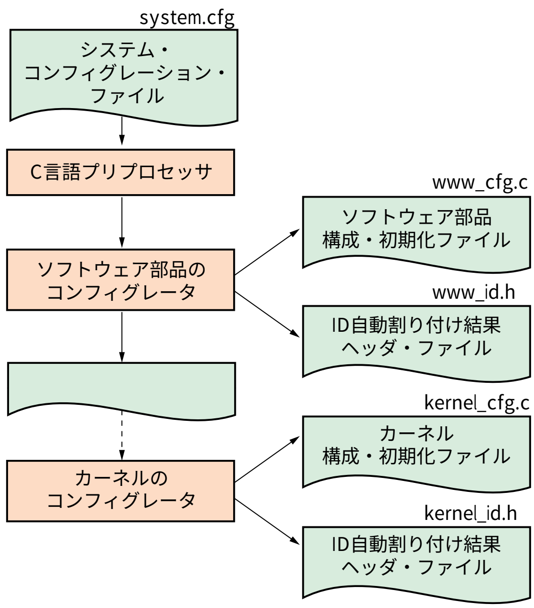


図6　ITRONのコンフィグレーションの図
仕様書からの抜粋．なお，図中のファイル名は例である．

ステム時刻に依存して行う処理としては次のようなものがあります．

- タイムアウトに伴う待ち状態からの解除
- dly_tsk（自タスクの遅延）による時間経過待ち状態からの解除
- 周期ハンドラの起動

タイムアウトに伴う待ち状態と時間経過待ち状態からの解除は，待ち状態を解除されるタスクが実行される（属する）プロセッサのシステム時刻に依存して行われます．周期ハンドラの起動は，周期ハンドラが実行される（属する）プロセッサのシステム時刻に依存して行われます．

● 複数のプロセッサの静的APIは一つのファイルに記述

μITRON 4.0仕様の標準プロファイルでは，オブジェクトの静的な生成のみをサポートしています．オブジェクトの登録（生成）情報は，静的APIと呼ばれるAPIによりコンフィグレーション・ファイルに記述します．コンフィグレーション・ファイルは，カーネル構築時にコンフィグレータによって処理され，カーネルの内部情報が生成されます（**図6**）．

機能分散型マルチプロセッサへの拡張仕様では，すべてのクラスはいずれかのプロセッサに属すると定義しているため，オブジェクトの登録などを行う静的APIは，次のようにクラスの囲みの中に記述します．

```
local_class <プロセッサ名>{
  そのプロセッサのクラスに属するオブジェクトの登録
}
```

つまり，複数のプロセッサの静的API記述を一つのファイルに記述するのです．

PE1，PE2という二つのプロセッサから構成されているシステムのコンフィグレーションの例を**リスト1**に示します．それぞれのプロセッサで二つのタスク（TASK1，TASK2）と一つの周期ハンドラ（CYCHDR1）を生成しています．

コンフィグレータは，プロセッサごとにプロセッサ名のディレクトリを作成し，そのディレクトリに各プロセッサのオブジェクト生成情報を記述したkernel_cfg.cと，オブジェクトの自動ID割り付けファイルであるkernel_id.h，およびクラスIDを付加したオブジェクトIDが記述されているclass_id.hを生成します．そして，プロセッサ名のディレクトリで，生成されたファイルとカーネルのソース・コードとユーザ・アプリケーション・ソース・コードをビルドして，プロセッサごとの実行オブジェクトを生成します．

リスト1　コンフィグレーション・ファイルの例

```
#ifndef _MACRO_ONLY
local_class PE1{
        INCLUDE("¥"sample1-dual1.h¥"");
        CRE_TSK(TASK1, { TA_HLNG, (VP_INT) 1, ..);
        CRE_TSK(TASK2, { TA_HLNG, (VP_INT) 2, ..);
        ...
        CRE_CYC(CYCHDR1, { TA_HLNG, 0, ..);
        #include "../systask/timer.cfg"
        ....
}
local_class PE2{
        INCLUDE("¥"sample1-dual2.h¥"");
        CRE_TSK(TASK1, { TA_HLNG, (VP_INT) 1, ..);
        CRE_TSK(TASK2, { TA_HLNG, (VP_INT) 2, ..);
        ...
        CRE_CYC(CYCHDR1, { TA_HLNG, 0, ..);
        #include "../systask/timer.cfg"
        ....
}
```

ほんだ・しんや
名古屋大学 情報連携基盤センター 名古屋大学組込みソフトウェア技術者人材養成プログラム

2